©2002 Microsoft Corporation

Object Oriented Concepts in .NET

Agenda: Day 1

Module 01: What is Microsoft® .NET?

Module 02: Visual Studio® .NET

Module 03: The .NET Languages

Module 04: Building ASP® .NET Applications

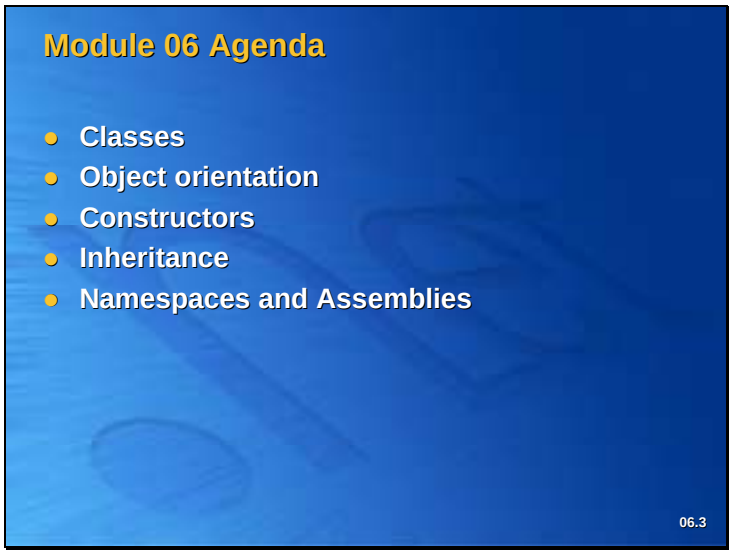
Module 05: Using Data Access in .NET Applications

▶ Module 06: Object Oriented Concepts in .NET

Module 07: Creating Windows Applications

Module 08: Using XML in .NET

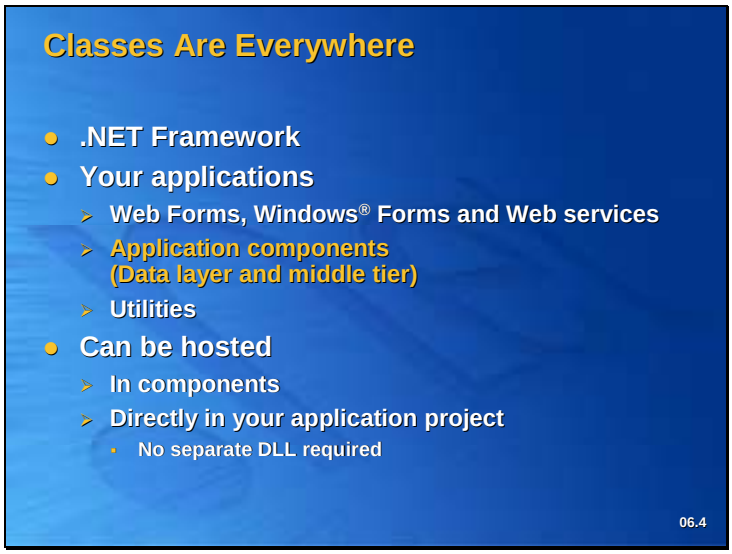
 ©2002 Microsoft Corporation



Module 06 Agenda

What you will learn:

- The role classes, constructors, inheritance, namespaces and assemblies play in achieving reusability for middle-tier components in the .NET Framework.



Classes Are Everywhere

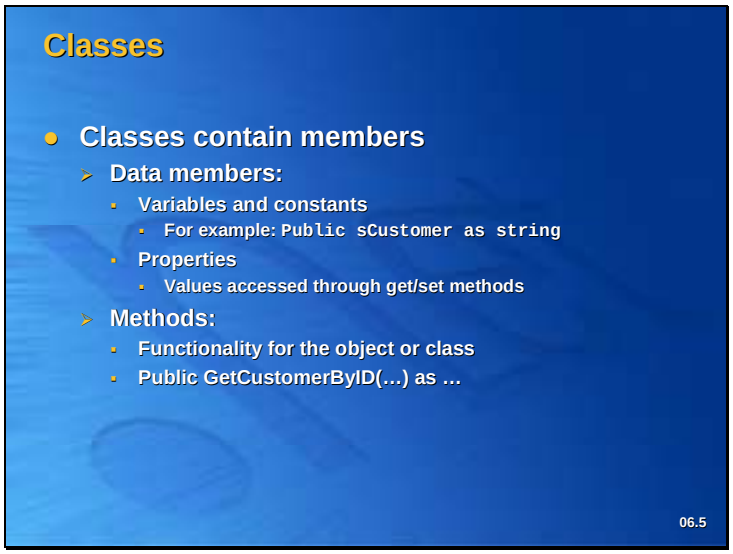
A class is a template for creating objects that contain both data and the methods that operate on that data. A class therefore defines a data type in terms of the operations that can be performed on it. Classes are a fundamental unit in .NET. Most of the types in .NET are classes. Classes are so fundamental that in most .NET languages you need to create a class just to write a simple “Hello World!” program.

The .NET Framework is more than a source for base classes. It ensures compatibility between classes created with different programming languages. Visual Basic® .NET classes can inherit from classes in assemblies created by any language that is .NET compatible.

The classes you create in Visual Basic .NET can also be used as base classes by other .NET programming languages. Visual Studio .Net applications can use many of the classes found in the .NET Framework as base classes.

Attribute classes, for instance, are always based on the class **System.Attribute**. Other useful .NET Framework base classes include **System.ComponentModel.Component**, which is inherited from to define components, and **System.Exception**, which is inherited from to define custom exception classes.

 ©2002 Microsoft Corporation



Classes

If you are familiar with object-oriented programming, you know that a class defines the operations an object can perform (methods, events, or properties) and defines a value that holds the state of the object (fields). Although a class generally includes both definition and implementation, it can have one or more members that have no implementation.

An instance of a .NET Framework class is an object. You access an object's functionality by calling its methods and accessing its properties, events, and fields. Each language chooses its own syntax for creating instances of classes.

Method - Describes operations available on the type. A method's signature specifies the allowable types of all its arguments and of its return value. A constructor is a special kind of method that initializes new instances of a type.

Property - Names a logical value or the state of the type and defines methods for getting or setting the property's value. Properties are often used to keep the public interface of a type independent from the type's actual representation.

Event - Defines an incident that can be responded to, and defines methods for subscribing to, unsubscribing from, and firing the event. Events are often used to inform other types of state changes.

Field - Describes and contains the value of a type. Fields can be of any type supported by the runtime.

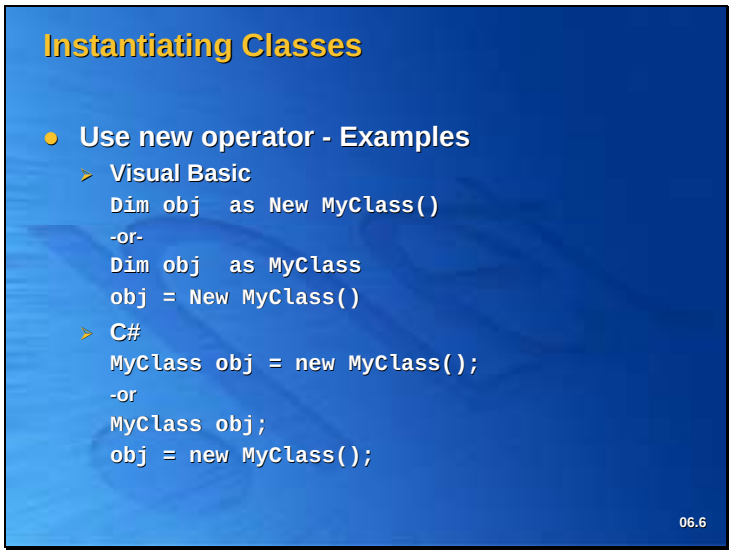
Nested type - Defines a type within the scope of the enclosing type.

Class members have characteristics. The common type system allows type members to have a variety of characteristics, but languages are not required to support all of these characteristics.

Members can either be data members or method members

Data members include fields (store instance or static data) and properties (convert syntax that looks like field access into a method call). Properties can be accessed by using the field-access syntax.

Function members include methods (instance or static functions that are part of the class) and events (handle the asynchronous events fired in a system).



Instantiating Classes

To declare a class, simply place the word “class” in front of the name of that class to create the template.

To use a class, you use the “new” operator to create an instance of that class.

An object is a typed region of storage in the execution environment; in addition to retaining state information, it also defines behavior. Class-type objects are defined using *class-name*. For example to define a class named “Account”:

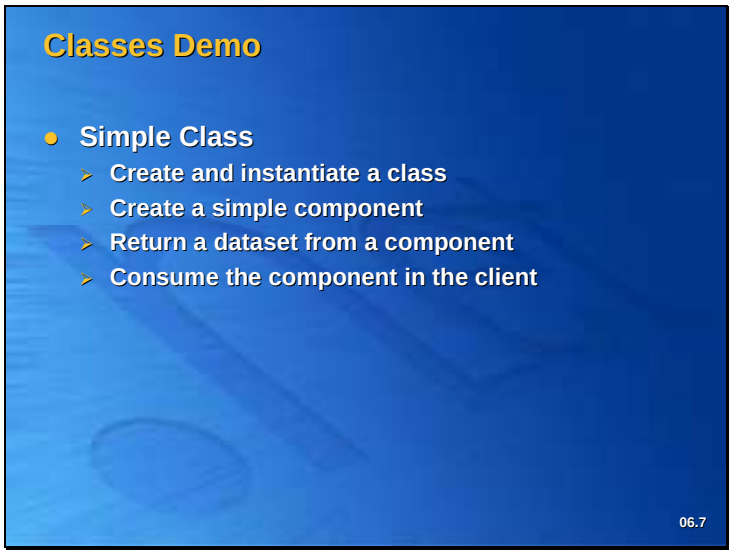
Visual C++

```
// class_type_objects.cpp
class Account
 {
 public:
      Account() // Default constructor
      {
       }
      Account(double); // Construct from double.
      double& Deposit(double);
      double& Withdraw(double, int);
 };
void main()
{
      Account CheckingAccount; // Define object of class type.
```

 ©2002 Microsoft Corporation

```
}
```

The code declares a class (a new type) called Account. It then uses this new type to define an object called CheckingAccount.



Classes Demo

Simple Class

The purpose of this demo is to show:

- How to create a simple component
- How to return a dataset from the component
- How to consume the component in a client application
- How the component can be used over and over by many applications

 ©2002 Microsoft Corporation

Object Orientation

Inheritance provides the mechanism by which a new class can “inherit” the behavior of an existing class and then is typically modified. For example, one could start with the “cat” class, which characterizes a cat as a mammal having four legs, etc., inherit from that class and add short fur and minimal meowing to get the “devon rex” class. (Devon Rex is a short-furred and mostly quiet breed of cat.)

Overloading Interfaces – The type of overloading supported across .NET CLS languages is **Method Parameter Overloading**.

Method Overloading is when you create multiple versions of a function that differ only by their parameter list. In a class, you could have

```
Public Function GetOrders() As DataSet
'Returns all orders
End Function Public Function GetOrders(ByVal CustomerId As Integer)
As DataSet
'Returns orders for a given customer
End Function
Public Function GetOrders(ByVal BeginDate As DateTime, ByVal EndDate As
DateTime) As DataSet
'Returns orders in a date range
End Function
```

Then clients for the class could call the GetOrders function and pass it one of the supported sets of parameters. Depending on the parameters passed in, the correct function will be executed.

Shared members - A class can contain shared methods and shared fields, in addition to instance methods and instance fields. A shared member differs from an instance member in that it can be

accessed without creating an instance from the class. A client can access a shared method simply by calling the shared method name together with the class name.

Interfaces are a powerful programming tool because they allow you to separate the definition of objects from their implementation, allowing your objects to evolve without risk of breaking existing applications. Both interfaces and class inheritance each have advantages and disadvantages, and you may end up using a combination of both in your projects.

Shared members (Visual Basic®) are properties, procedures, and fields that are shared by all instances of the class. Some programming languages refer to such items as *static* or *class* members. Sharing a single instance of a data member or function among all instances of a class is useful in Visual Basic applications that use inheritance. Shared data members exist independently of any particular instance of a class. Shared procedures are not implicitly passed instances of the class. For this reason, no unqualified references to non-shared data members are allowed in shared methods. Public shared members can be accessed remotely and can be late-bound from instances of the class.

Note These are called *static* or *class* members in some other languages.

Constructors and Finalizers – .Net does not have Destructors. In the garbage collected world, we have Finalizers that are run when the garbage collector gets around to cleaning up released references. Classes can also have an optional Dispose method for clients to call to free up resources before releasing the reference. See the Glossary description for Deterministic Finalization for a description of the objection destruction process in the .NET garbage collector.

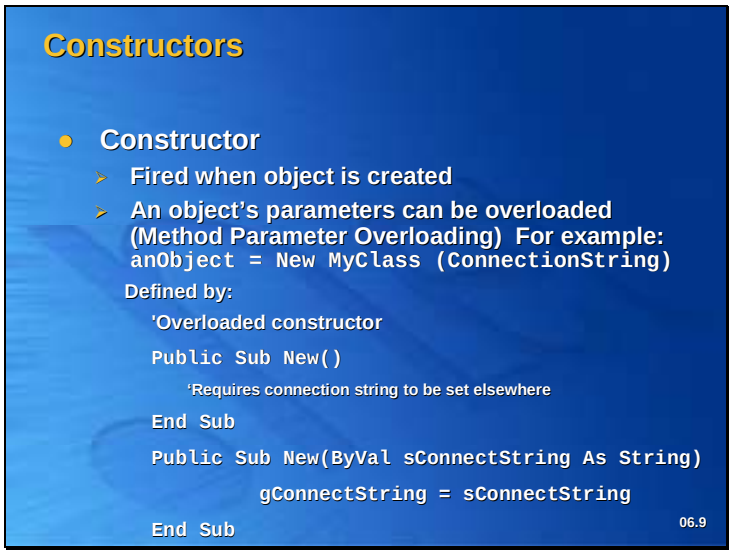
Objects begin life when an instance of a class is created using the **New** keyword. Microsoft Visual Basic .NET controls the initialization of new objects using procedures called **constructors**. Common initialization tasks include opening files, connecting to a database, and reading the values of registry keys. Objects end their life after they leave scope or are set to **Nothing** and are released by the .NET framework. Visual Basic .Net controls the release of system resources using procedures called **Finalizers**. Used together, constructors and Finalizers support the creation of robust and predictable class libraries.

Events and delegates are closely associated in Visual Basic .NET. An **event** is a message sent by an object announcing that something important has happened. Events are implemented using delegates, a form of object-oriented function pointer that allows a function to be invoked indirectly by way of a reference to the function.

Delegates serve a similar purpose to function pointers in Microsoft Visual C++®. They are type safe, secure, managed objects that always point to a valid object and cannot corrupt the memory of another object. Unlike function pointers in Visual C++, delegates can reference static, virtual and instance methods. In addition to their use as the equivalent of function pointers, delegates are used for event handling and callbacks in the .NET Framework.

All delegates inherit from System.Delegate and have an invocation list, or a linked list of methods that are executed when the delegate is invoked. Each delegate declares a method signature and the resulting delegate can reference any method with a matching signature. The delegate declaration lists the number, order and type of parameters the delegate expects and a return type. Each instance of a delegate can forward a call to one or more methods that take those parameters and return the same type. The return value is not defined for a delegate that has a return type and contains more than one method in its invocation list.

 ©2002 Microsoft Corporation



Constructors

Visual Basic .NET uses **Sub New** as the constructor. Visual C#™ uses a function with the same name as the class.

Visual Basic .NET revises object creation for improved readability and to support **constructors** with arguments.

Visual Basic 6.0

In Visual Basic 6.0, an object variable declared with the keywords **As New** is initialized to **Nothing**, meaning that no object has yet been created. Every time the variable is encountered during execution, it is evaluated before the code executes. If the variable contains **Nothing**, an object of the appropriate class is created before the code that uses it executes.

Visual Basic .NET

In Visual Basic .NET, there is no implicit object creation. If an object variable contains **Nothing** when it is encountered, it is left unchanged and no instance is automatically created.

You can create an object with the same statement that declares the object variable. The following two lines of code are treated as equivalent, with the first being taken as shorthand for the second:

```
Dim Emp As New EmpObj() ' Shorthand for next line.
Dim Emp As EmpObj = New EmpObj() ' EmpObj created; Emp points to EmpObj.
```

In both cases, an EmpObj object is created as soon as the **Dim** statement is executed, and Emp is initialized to a reference to the new object.

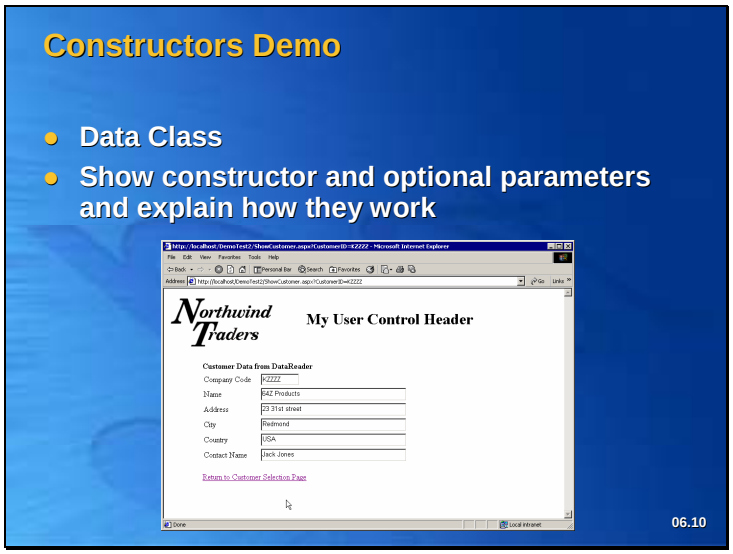
Some classes have constructors that take arguments. If you are creating an object from such a class, you can include its arguments in the declaration. The following example shows two equivalent declarations that pass arguments to the object's constructor, with the first being taken as shorthand for the second:

```
Dim Q As New Quark(12, 0.0035) ' Shorthand for next line.
Dim Q As Quark = New Quark(12, 0.0035) ' Parameterized constructor
```

The **Sub New** and **Sub Finalize** procedures in Visual Basic .NET replace the **Class_Initialize** and **Class_Terminate** methods used in previous versions of Visual Basic to initialize and destroy objects. Unlike **Class_Initialize**, the **Sub New** constructor will run only once when a class is created, and it cannot be called explicitly anywhere other than in the first line of code in another constructor from either the same class or from a derived class. Furthermore, the code in the **Sub New** method always will run before any other code in a class.

The **Finalize** destructor is a protected method that can be called only from the class it belongs to or from derived classes. **Finalize** is called automatically by the system when an object is destroyed, so you should not explicitly call **Finalize** from outside of a derived class' **Finalize** implementation or the **Finalize** destructor will be called more than once. If your class needs to allow clients to request that it release resources, it must implement the **IDisposable** interface that contains the **Dispose** method.

 ©2002 Microsoft Corporation



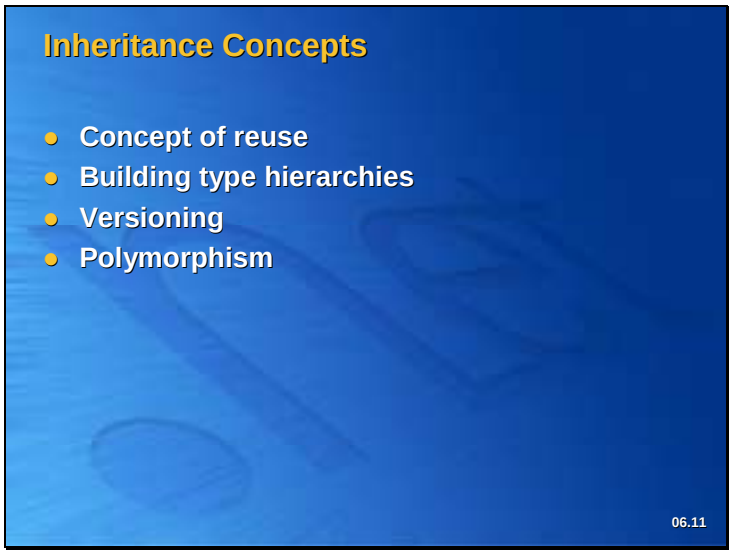
Constructors Demo

Data Class

Show how to create a new class that has a constructor and a richer set of methods.

The purpose of this demo is to illustrate:

- Show constructor and optional parameters and explain how they work.
- How data reader can be returned
- Added reuse of components



Inheritance Concepts

Well-designed classes can be debugged once and reused as a basis for new classes. Inheritance lets you reuse tested code in your objects. Almost everything you do with Visual Basic .NET involves objects, including some entities not normally thought of as objects such as strings and integers. All objects in Visual Basic .NET are derived from and inherit methods from a base class called **System.Object**. The **System.Object** class provides a common base class for all objects in the .NET framework, which in turn ensures interoperability of objects developed using Visual Studio .NET. New classes implicitly inherit the **System.Object** class. It is unnecessary to explicitly name this class with an Inherits statement.

Among the methods that objects inherit from **System.Object**, one of the most useful is **Object.GetType**, which is used to return the exact type of the current object.

Inheritance and “IS-A" Relationships

Two ways to show class relationships in object-oriented programming are “IS-A" and “HAS-A" relationships. In an “IS-A" relationship, the derived class is clearly a kind of the base class. For example, a class named PreferredCustomer represents an “IS-A" relationship with a base class named Customer because a preferred customer is a customer. However, a class named CustomerReferral represents a “HAS-A" relationship with the Customer class because a customer referral has a customer, but is not a kind of customer.

Objects in an inheritance hierarchy should have an “IS-A" relationship with their base class because they inherit the fields, properties, methods and events defined in the base class. Classes that represent a “HAS-A" relationship with other classes are unsuited to inheritance hierarchies because they may inherit inappropriate properties and methods. For example, if the CustomerReferral class was derived from the Customer class discussed previously, it might inherit properties that make no sense such as AverageBalance. "HAS-A" relationships such as this should be represented using unrelated classes or interfaces.

Even well-designed class **hierarchies** need to evolve over time. The choices you make early on when designing a class hierarchy can simplify things later. Here is a list things to take into consideration while designing your class hierarchies:

 ©2002 Microsoft Corporation

Class hierarchies are defined from the general to the specific. Define classes at each level of an inheritance hierarchy to be as general as possible. Derived classes inherit, reuse and extend methods from base classes. For example, if you are designing a class hierarchy that models computer hardware, when you start modeling output devices, you could define classes named Display, Printer and File. You could then define the classes that implement the methods defined in the base classes. For example, the LCDDisplay class could be derived from the Display class and implement a method named EnterPowerSaveMode.

Be generous in defining data types and storage to avoid difficult changes later. For example, you might consider using a variable of type **Long** even though your current data may only require a standard **Integer** variable.

Only expose items you are sure are necessary, to derived classes. **Private** fields and methods reduce naming conflicts and prevent other users from using items you may need to change down the road.

Members that are required by derived classes should be marked as protected. This reduces the number of classes that are dependent on these members to just the derived classes making it easier to update those members during development of the class.

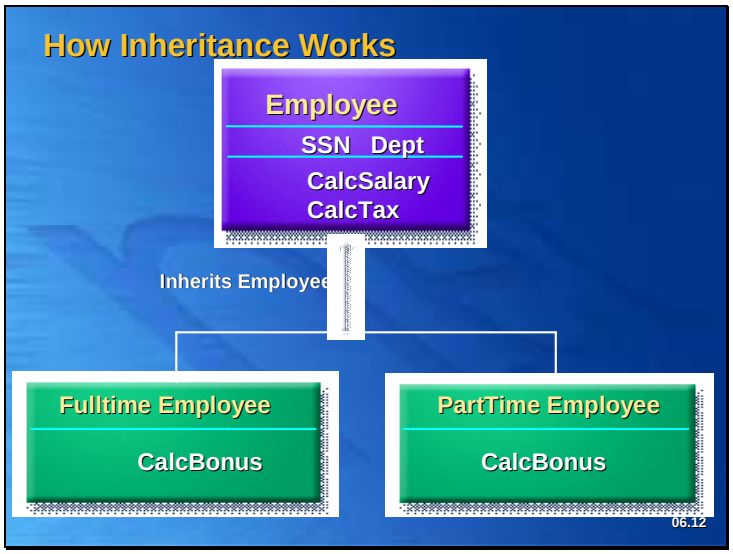
Versions of base classes must be carefully controlled so as to not break derived classes. Just like COM or other object-oriented classes, updates to base classes must not break interfaces that are used in derived classes or in consumers of those classes. Doing so may break all derived classes and/or consuming classes.

Polymorphism

Derived classes in a class hierarchy can sometimes be used interchangeably with their base class, a process called inheritance-based polymorphism. This approach combines the best features of interface-based polymorphism with the option of reusing or overriding code from a base class.

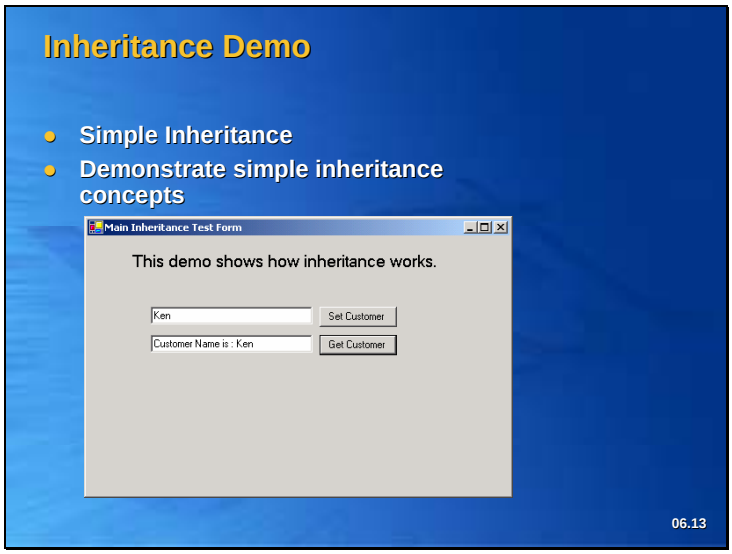
Visual C# Inheritance

Unlike Visual C++, only single inheritance is allowed in Visual C#. In other words, a class can inherit implementation from only one base class. However, a class can implement more than one interface.



How Inheritance Works

 ©2002 Microsoft Corporation



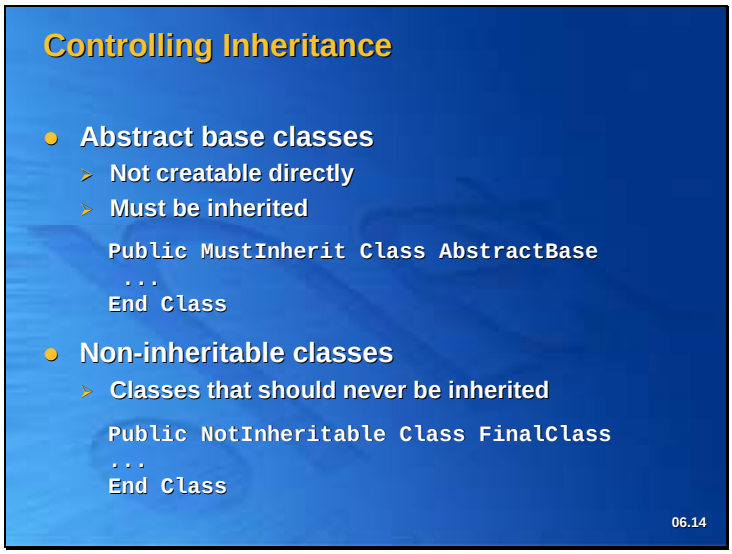
Inheritance Demo

Simple Inheritance

Demonstrate simple inheritance concepts

The purpose of this demo is to illustrate:

- How to create a base class?
- How to derive a class from it?



Controlling Inheritance

Abstract classes are closely related to interfaces. Abstract classes are classes that cannot be instantiated and are frequently either partially implemented or never implemented. One key difference between abstract classes and interfaces is that a class may implement an unlimited number of interfaces, but may inherit from only one abstract, or any other kind of, class. A class that is derived from an abstract class may still implement interfaces. Abstract classes are useful when creating components because they allow you to specify an invariant level of functionality in some methods, but leave the implementation of other methods until a specific implementation of that class is needed. This also helps in better version control because if additional functionality is needed in derived classes, it can be added to the base class without breaking code.

An abstract class is denoted in Visual Basic® by the keyword **MustInherit**. In Visual C#®, the **abstract** modifier is used. Any methods that are meant to be invariant may be coded into the base class, but any methods that are to be implemented are marked in Visual Basic with the **MustOverride** modifier. In Visual C#, the methods are marked **abstract**, as shown in the following example:

```
public abstract class Class1
{
      public Class1()
      {
            …
      }
      public abstract void method()
      {
            …
      }
```

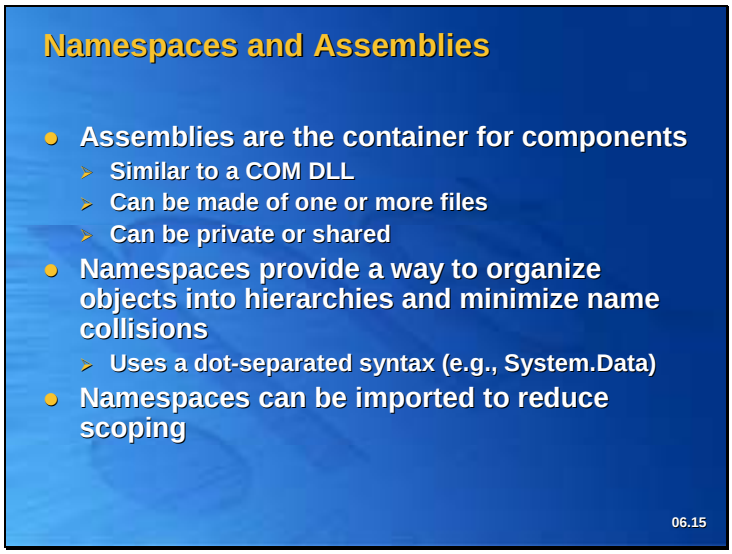
 ©2002 Microsoft Corporation

```
}
public class Class2     : Class1
{
      public override void method()
      {
            …
      }
}
```

Often, one must decide whether to use an interface or an abstract class to describe a set of methods and properties to be implemented in inherited classes. A simple resolution is the “IS-A" rule: If the essential type of the derived classes "IS-A" specialization of the base type, then inherit from an abstract class. Use an interface if the interface is not related to the essential type of the actual objects because interfaces express what an object "CAN-DO," not what it is.

Be careful when you design your interfaces. Once you've published them, you shouldn't change them, even to add a method. (If you add a method, every class that implements your interface also will have to implement that method and they'll be broken until they do.) You can add methods, however, without breaking derived classes.

The Visual Basic .NET, NotInheritable keyword prevents derivation from a class. In C# the equivalent keyword is sealed.


Namespaces and Assemblies
Assemblies are the container for components
Similar to a COM DLL
Can be made of one or more files
Can be private or shared
Namespaces provide a way to organize objects into hierarchies and minimize name collisions
Uses a dot-separated syntax (e.g., System.Data)
Namespaces can be imported to reduce scoping
06.15

Namespaces and Assemblies

Managed DLLs (Assemblies)

The functionality of the .NET Framework class library is not contained in a single DLL. By putting the functionality of the Framework in multiple DLLs, a managed program does not need to load, at start up, a large DLL, but, rather, one or more smaller DLLs. This reduces the start up time for a program.

You can use namespaces that are defined in your project. However, you will commonly use types in namespaces that reside in a managed DLL. A managed DLL is also called an assembly.

When you create a Visual Basic or Visual C# project in Visual Studio, the most common assemblies are already referenced. However, if you need to use a type that is in an assembly that is not already referenced, you will need to add a reference to the assembly. The Add Reference dialog box allows you to add a reference to an assembly.

Namespaces

The .NET Framework class library is comprised of **namespaces**. Each namespace contains types you can use in your program: classes, structures, enumerations, delegates and interfaces. For an overview of namespaces, see .NET Framework Class Library in the Visual Studio .NET documentation.

Creating Namespaces: Both Visual Basic and Visual C# use the namespace statement to create a new namespace. All namespaces shipped by Microsoft will begin with one of two names:

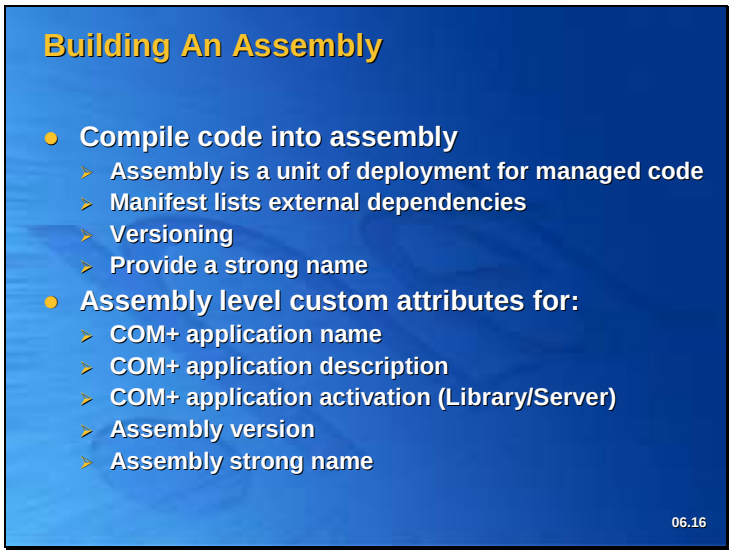
System - Namespaces developed by the Microsoft .NET Framework SDK team.

Microsoft® - Namespaces developed by other groups at Microsoft to offer their user features that can be used in applications that target the Common Language Runtime.

Namespaces provide **scope**: Two classes with the same name can be used in your program as long as they are in different namespaces and as long as you qualify the names to the namespaces. The namespace name is part of the fully qualified name of the type (*namespace*.*typename*). If you do not want to use fully qualified names (*namespace*.*typename*), you can use the **Imports statement** in Visual Basic or the **using keyword** in Visual C# or the **Imports directive** in ASP.

 ©2002 Microsoft Corporation

The **scope** of the elements made available by an imports statement depends on how specific you are when using the imports statement. For example, if only a namespace is specified, all uniquely named members of that namespace are available without qualification. If both a namespace and the name of an element of the namespace are specified, only the element or members of the element are available.



Building an Assembly

In general, an assembly can consist of four elements:

- The assembly manifest, which contains assembly metadata
- Type metadata
- Microsoft Intermediate Language (MSIL) code that implements the types
- A set of resources

Although only the assembly manifest is required, types or resources are necessary to give the assembly any meaningful functionality. The elements of an assembly can be contained in one or more files.

The Manifest is the reason why the registry is unnecessary to the assemblies. It contains all the information that the CLR needs to execute (manage) it.

Every assembly contains a collection of data that describes how the elements in the assembly relate. An assembly manifest contains all the metadata needed to specify the assembly's version requirements and security identity as well as all metadata necessary to define the scope of the assembly and resolve references to resources and classes. The assembly manifest can be stored in either a portable executable (PE) file (an .exe or .DLL) with Microsoft Intermediate Language (MSIL) code or in a PE file that contains only assembly manifest information.

Assembly Manifest Contents

The following list shows the information contained in the assembly manifest. The first four items — the assembly name, version number, culture and strong name information — make up the assembly's identity.

Assembly name - A text string specifying the assembly's name.

Version number - A major and minor version number as well as a revision and build number. The CLR uses these numbers to enforce version policy.

 ©2002 Microsoft Corporation

Culture - Information on the culture or language the assembly supports. This information should only be used to designate an assembly as a satellite assembly containing culture- or language-specific information. An assembly with culture information is automatically assumed to be a satellite assembly.

Strong name information - If the assembly has been given a strong name, the manifest contains the public key from the publisher.

List of all files in the assembly - A hash of each file contained in the assembly and a file name.

Type reference information - Information used by the runtime to map a type reference to the file that contains its declaration and implementation. This is used for types that are exported from the assembly.

Information on referenced assemblies - A list of other assemblies that are statically referenced by the assembly. Each reference includes the dependent assembly's name, assembly metadata (version, culture, operating system and so on) and public key, if the assembly is strong named.

All **versioning** of assemblies that use the Common Language Runtime is done at the assembly level. The specific version of an assembly and the versions of dependent assemblies are recorded in the assembly's manifest. The default version policy for the CLR is that applications run only with the versions they were built and tested with, unless overridden by explicit version policy in configuration files (the application configuration file, the publisher policy file, and the computer's administrator configuration file).

A **strong name** consists of a public key and a digital signature in addition to the simple text assembly name. It is generated over an assembly file (the file that contains the assembly manifest, which in turn contains the names and hashes of all of the files that make up the assembly) using a corresponding private key. Signing an assembly with a strong name allows the assembly to be deployed in the global assembly cache.

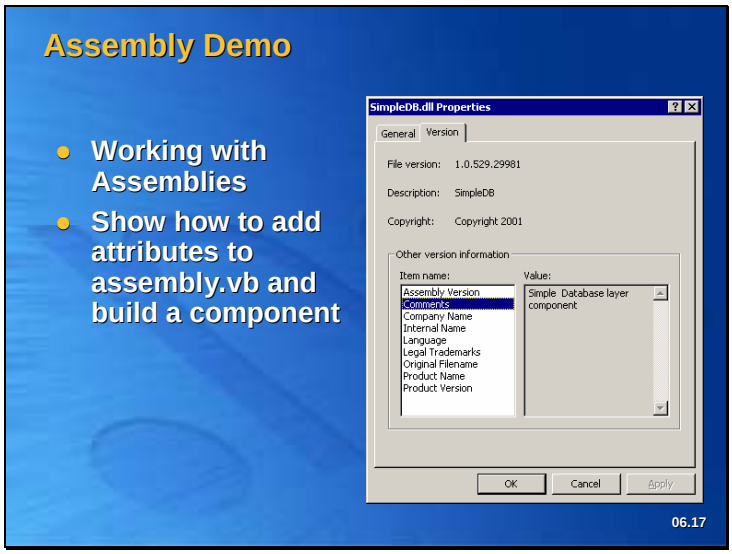
Assembly Identity Attributes

Assembly attributes fall into one of these four categories:

- Assembly identity attributes
- Informational attributes
- Assembly manifest attributes
- Strong name attributes

The name, version and culture attributes, together with a strong name (if applicable), determine the identity of an assembly. These attributes form the full name of the assembly and are required when referencing the assembly. You can use attributes to set an assembly's version and culture. The compiler or the Assembly Generation tool (Al.exe) sets the name value when the assembly is created based on the file containing the assembly manifest.

For more information on accessing COM+ services from .Net components, see .Net Framework SDK Documentation on **System.EnterpriseServices Namespace**



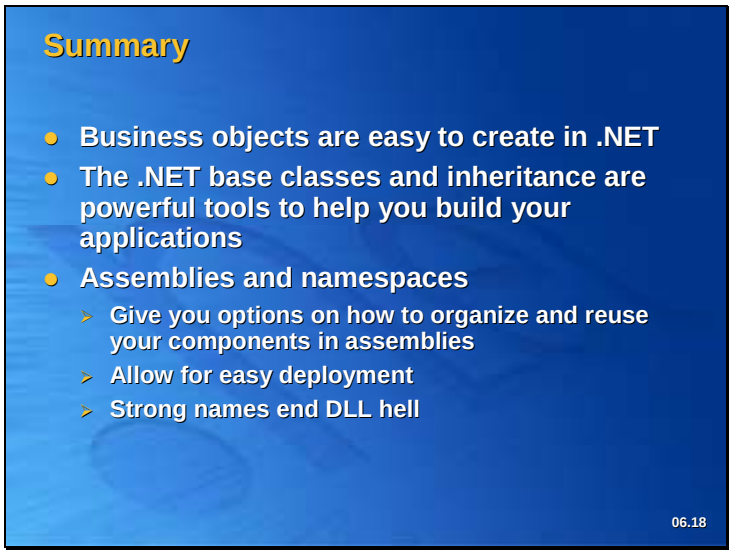
Assembly Demo

Working with Assemblies

The purpose of this demo is to illustrate:

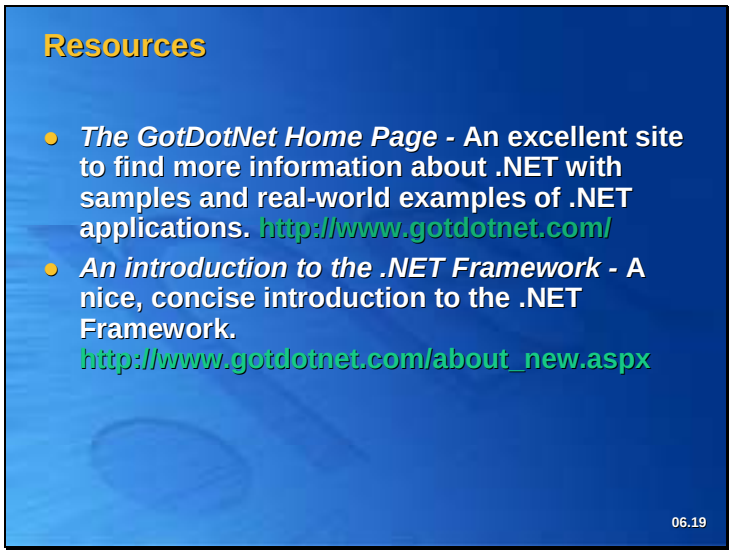
- How to set assembly attributes?
- How to set build options and build release code?
- How to view properties with Windows Explorer?

 ©2002 Microsoft Corporation


Summary
Business objects are easy to create in .NET
The .NET base classes and inheritance are powerful tools to help you build your applications
Assemblies and namespaces
Give you options on how to organize and reuse your components in assemblies
Allow for easy deployment
Strong names end DLL hell
06.18

Summary

The .NET Framework is built on classes and object-oriented methodologies that promote reuse of your code through inheritance.


Resources
The GotDotNet Home Page - An excellent site to find more information about .NET with samples and real-world examples of .NET applications. http://www.gotdotnet.com/
An introduction to the .NET Framework - A nice, concise introduction to the .NET Framework. http://www.gotdotnet.com/about_new.aspx
06.19

Resources

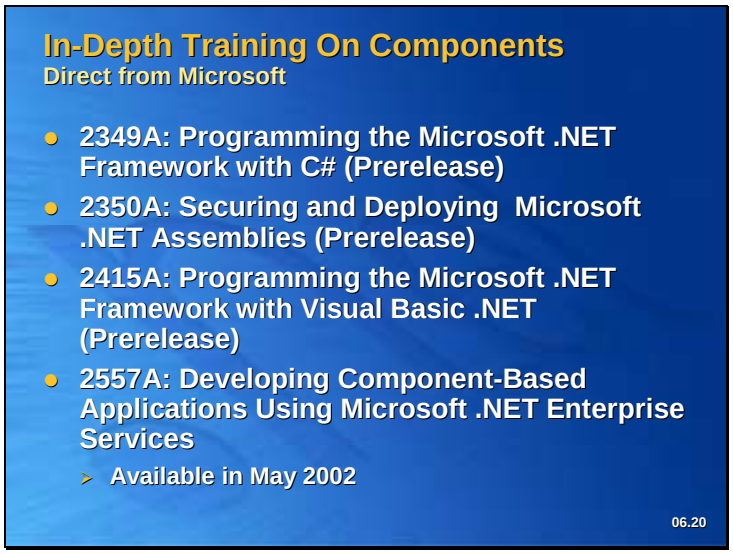
The GotDotNet Home Page - An excellent site to find information on .NET with samples and real-world examples of .NET applications.

http://www.gotdotnet.com/

An introduction to the .NET Framework - A nice, concise introduction to the .NET Framework.

http://www.gotdotnet.com/about_new.aspx

 ©2002 Microsoft Corporation


In-Depth Training On Components
Direct from Microsoft
2349A: Programming the Microsoft .NET Framework with C# (Prerelease)
2350A: Securing and Deploying Microsoft .NET Assemblies (Prerelease)
2415A: Programming the Microsoft .NET Framework with Visual Basic .NET (Prerelease)
2557A: Developing Component-Based Applications Using Microsoft .NET Enterprise Services
Available in May 2002
06.20

In-Depth Training On Components

Direct from Microsoft

 ©2002 Microsoft Corporation